Introduction to high performance computing

Cecina Babich Morrow

2024-02-23

The following portfolio contains materials covered in ACRC's Introduction to High Performance Computing.

What is HPC?

High performance computing (HPC) is the use of combined computational power of many individual computers via supercomputers or clusters. The following diagram shows the architecture of an HPC system, including both login and compute nodes:

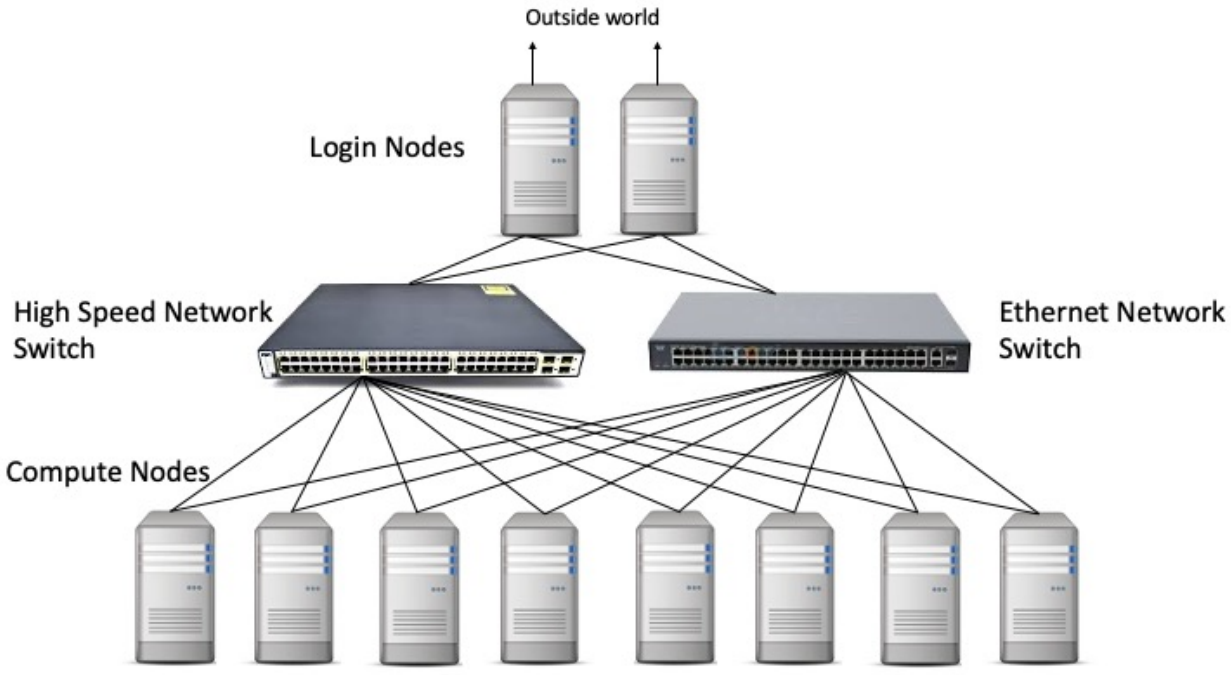

Figure 1: Diagram from ACRC

When running tasks on a supercomputer, you prepare your jobs on the login node and submit the jobs to the compute nodes. It is important to note that you should not run jobs on the login nodes. The following diagram shows the workflow for running a job on a supercomputer:

First, you must log in to the login node using SSH. Then, you prepare your job and submit it to the scheduler (SLURM in our case). The scheduler then allocates resources and runs the job on the compute nodes. Once the job is complete, the results are returned to the login node. You can then view and analyse the results, making sure to download / back up any important files.

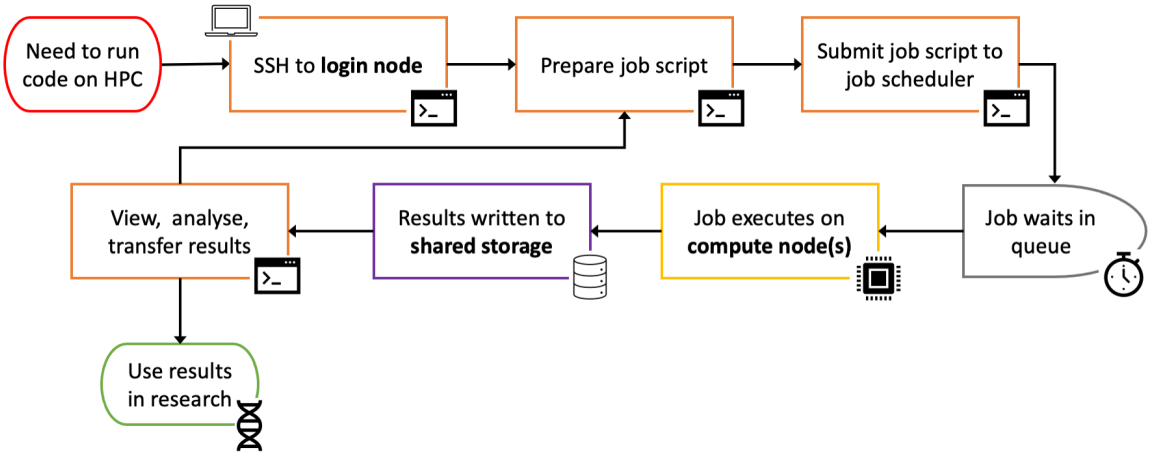

Figure 2: Diagram from ACRC

HPC at University of Bristol

At the University of Bristol, we have access to two clusters: BlueCrystal and BluePebble. BlueCrystal Phase 4 is the older of the two and is recommended for most workloads, while BluePebble is recommended for high throughput computing. We can also access regional and national computing facilities including ARCHER2 and Isambard.

It is important to remember that nothing on the HPC is backed up, so you should always ensure that you back up your data. We have access to the following locations for data storage:

- home directory: fairly small
- scratch space: 1 TB to store the data (not backed up)
- RDSF for longer term storage

For any issues with HPC at University of Bristol, you can contact the following:

- hpc-help@bristol.ac.uk for any help with HPC
- rdsf-help@bristol.ac.uk for data storage help
- ask-rse@bristol.ac.uk for help writing software

For the rest of this portfolio, we will be using BlueCrystal.

Logging in

We log into clusters using `ssh`, which stands for "secure shell" and is a command-line utility for securely logging into a remote machine and executing commands on that machine. The syntax is `ssh [username]@[hostname]`.

For example, in order for me to log in to BlueCrystal, I run `ssh aw23877@bc4login.acrc.bris.ac.uk`, which lands me in my home directory `/user/home/aw23877`.

Writing shell scripts

Job shell scripts start with `#!/bin/bash` and typically have the file extension `.sh` (you can also use `.slm` for Slurm as an alternative).

The following is a very simple shell script:

```
#!/bin/bash
#

#SBATCH --partition=test
#SBATCH --account=MATH030984

# Change into working directory
#cd ~/workshop

# Execute code
/bin/hostname

# Pause to give us time to see the job
sleep 60
```

Some things to note about this script:

- `#SBATCH --partition-test` tells us which partition to use, since the cluster is broken up into partitions for different purposes
  - `test` can be used to test out if jobs work
- `#SBATCH --account=MATH030984` uses your account number – you might be associated with multiple projects over time, this is for reporting purposes
- `hostname` tells you which node you are on

Here is a slightly more complex shell script:

```
#!/bin/bash
#
#
#SBATCH --partition=teach_cpu
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --time=0:0:10
#SBATCH --mem=100M
#SBATCH --account=MATH030984

# Define executable
export EXE=/bin/hostname

# Change into working directory
cd "${SLURM_SUBMIT_DIR}"

# Execute code
${EXE}
```

We now have more SBATCH directives at the start of our script:

- `--nodes` is the number of nodes you want to run on (almost always one node is sufficient)
- `--ntasks-per-node` is the number of tasks you want to run on each node
- `--cpus-per-task` is the number of CPUs you want to use for each task (if your code can use multiple threads, you can set that here)
- `--time` is the maximum time you want to run for, it's used to balance out the load on the cluster (the shorter this is, the sooner your job will run, but if you specify a time shorter than how long your code will take, it will kill your job)
- `--mem` is the maximum memory you want to use. you can perform experiments to understand how much time and memory your code will take to run

In the rest of the script, we start by exporting the variable `EXE`, which is set to `/bin/hostname`. To access variables in bash, you use `${}`, either with or without quotes (`"${SLURM_SUBMIT_DIR}"` or `${EXE}`).

Slurm

Slurm is the job scheduler. All Slurm commands start with `s`. In order to submit a job, we use the command `sbatch`. We are using a batch supercomputer, which means that jobs are submitted and then run when resources are available. To run our shell script `job1.sh`, we can run:

```
sbatch job1.sh
```

This command returns `Submitted batch job 12343792`, for example. That number is the job id. It is best practice to keep track of your job ids and make note of why you submitted them in case you need to revisit them at a later date.

Running `sacct` will show you the status of all your jobs, for example:

```
JobID           JobName  Partition    Account  AllocCPUS      State ExitCode
------------ ---------- ---------- ---------- ---------- ---------- --------
12343792        job1.sh       test math030984          0    PENDING      0:0
```

This output shows the job ID, job name (can be set using `#SBATCH --job-name "your job name"`), what partition you are running on, and what account you are using. Once the job is actually running (when State is no longer `PENDING`), it will tell you how many CPUs are allocated. It tells you the job state (pending, running, completed, etc.) and the exit code (0 means it ran successfully).

Sometimes jobs will be displayed on multiple lines when running `sacct`, so you can add the flag `sacct -X` shows you the status of your jobs with just one line per job.

As an alternative to `sacct`, `squeue` shows the status of all jobs on the cluster. `squeue --me` shows the status of just your jobs and tells you why something hasn't run yet.

If your job ran successfully, you will end up with an output file like `slurm-12343792.out` (with that job id number you noted earlier) that includes what would have been printed out if the job was run on your computer.

Running `sinfo` tells you which nodes are available and how many CPUs are available on each node.

Other helpful commands

Here are some other helpful commands for working with the supercomputer:

- `user-quota` tells you how much disk space you have used up and what the limits are
- `module avail` prints out all of the available modules – note that when you log out and log back in, you have to reload all of the modules you need again
- `module spider anaconda` searches for anaconda in all of the available modules
- `module load languages/anaconda3` loads the anaconda module (`module unload languages/anaconda3` unloads it)
- `module list` shows you what modules you have loaded currently

Copying files to and from the supercomputer

It is often necessary to move local files from your computer onto the supercomputer and vice versa. There are a few ways to do this. If you are using Git for version control, you can use `git` to clone a repository, push, and pull as normal. If not, you can use `scp` or `rsync`.

`scp` stands for "secure copy" and is a command-line utility for securely copying files between two locations. The syntax is `scp [source] [destination]`. To copy a file from the supercomputer to your machine, for example, `scp aw23877@bc4login.acrc.bris.ac.uk:workshop/script.py .` copies `script.py` from the `workshop` directory on the supercomputer to the current directory on your local machine. Conversely, `scp ./meaning_of_life.txt aw23877@bc4login.acrc.bris.ac.uk:workshop` copies `meaning_of_life.txt` from your local machine to the `workshop` directory on the supercomputer.

As an alternative to `scp`, `rsync` is a more powerful version of `scp` that allows you to copy entire directories.

Simplifying logging in

You can make it easier to log in by editing your `~/.ssh/config` file to specify the host name and your username so you don't have to type these in every time you want to use the super computer. For example, you can add the following to your `~/.ssh/config` file:

```
Host *
        AddKeysToAgent yes

Host bc4
        HostName bc4login.acrc.bris.ac.uk
        User aw23877
```

Now you can just run `ssh bc4` to log into BlueCrystal and `scp file-name bc4:workshop` for copying a file to the supercomputer.

At this point, you still need to provide your password every time. To work around that, you can use SSH keys. `ssh-keygen -t ed25519` generates a new SSH key. You should add a passphrase to your SSH key in case someone else gets access to your laptop. Running `ssh-keygen -t ed25519` and adding a passphrase will write two files: `id_ed25519` is the private key and `id_ed25519.pub` is the public key. `ssh-copy-id bc4` copies your public key to the supercomputer so you don't have to type in your password every time you log in (you only need to run this once per computer).

Now you can run `ssh bc4` without entering your password (you will need to enter the passphrase for your SSH key when you restart your computer, but then you can log in without needing to re-enter any password).

Array jobs

Array jobs are a way to run the same job many times with different inputs. This is useful for running the same analysis on different data sets, for example.

Let's start with a simple python script `script.py`. We can edit `script.py` to take in an argument:

```
import sys

data = sys.argv[1]

print(f"Running analysis {data}")
```

This allows us to pass in a different argument to the script each time we run it, e.g. `python3 script.py 42` will print out `Running analysis 42`.

We can now edit our shell script to run our Python script with different arguments each time. We can use the `--array` flag to specify how many times we want to run the job and what the arguments should be. Here is an example of a shell script that runs `script.py` 5 times with different arguments each time:

```
#!/bin/bash
#
#SBATCH --partition=teach_cpu
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --time=0:0:10
#SBATCH --mem=100M
#SBATCH --account=MATH030984
#SBATCH --job-name="python test"
#SBATCH --array=1-5

module load languages/anaconda3

python3 script.py "${SLURM_ARRAY_TASK_ID}"
```

`--array=1-5` tells Slurm to run the job 5 times with the array task ID set to 1, 2, 3, 4, and 5.